

# INTRODUCTION

Programming is about the creation of software to solve problems. Problems come in many forms: simple to complex, small to large, I/O-intensive to compute-intensive. Over the past four decades, we've tried to solve a lot of different types of problems with software. At some, we have been exceptionally successful and the solution space is well understood. For others, such as those proposed by artificial intelligence, we have only been marginally successful (and, in many cases, not at all) within limited domains.

But, programming has always been both a knowledge-intensive and labor-intensive process. As PCs became more pervasive and the demand for applications increased, a problem arose—how to develop safe, robust software quicker.

Traditionally, the software industry has focused on delivering high-quality, special purpose application software to the end-user and high-quality, general purpose development software to the professional developer. End-users are not expected to develop their own applications. Much effort has been invested in making software development more efficient for professional developers by providing huge collections of prefabricated software components. Much effort has also been invested in increasing the utility of application software by providing customizable user interfaces.

Most software development tools are too complex for end-users. The few that are simple enough to attract nonprogrammers limit the user to simple applications that can't be combined with other software. Applications have to be built from scratch; they do not scale and, therefore, are often of limited practical use. On the other hand, commercial application software offers little modifiability beyond customization of

## 2 INTRODUCTION

the user interface. In particular, applications do not let users modify their behavior or reuse their functionality within other applications.

The solution to this problem requires the following assumptions:

1. High-quality software applications should be producible with some skill and effort by taking advantage of functionality in software components developed by others.
2. Flexible integration of new software applications should be achievable by decomposing applications into small functional components that can be reused independently and recombined efficiently.

The first assumption requires that users focus more on gluing components together rather than developing individual components, although they will have to do some of the latter. The second assumption requires that users have standard ways of describing and representing components as well as guides for how to connect those components together.

Software developers and users should be able to build small applications fast and easily. The learning effort required to start building an application should stay in the range of hours. Nevertheless, users should expect their applications to be comparable in look and feel to commercial software.

To achieve this goal, there are four problems to be addressed:

1. How do we build more complex structures from simple parts?
2. What is the best way to integrate (“glue together”) multiple parts to form a whole?
3. How can we construct parts to make them reusable, and reuse them?
4. How can we ensure that small solutions can scale to bigger problems?

This book will provide some answers to these questions. We do so through examination of different conceptual paradigms—from the small to the large for constructing software applications.

### 1.1 THE MEANING OF PARADIGM

*Paradigm* (a Greek word meaning example) is commonly used to refer to a category of entities that share a common characteristic. Numerous authors have examined the concept of a paradigm. Perhaps the foremost user of the word has been Thomas Kuhn (1996), who wrote the seminal book *The Structure of Scientific Revolutions*.

Kuhn used the notion of paradigm in the scientific process by defining it as a scientist’s view of the world and the structure of his assumptions and theories, which affect that view. In his definition, he included the concepts of law, theory, application, and instrumentation: in effect, both the theoretical and the practical.

Kuhn saw a “paradigm” as emerging as the result of social processes in which people develop new ideas and create principles and practices that embody these ideas. Large software development is a social process, because it is often a team effort. Each new software application is a unique creation in its own right. Rarely, if ever, does a team of programmers set out to create a program that exactly mimics the code and structure of some other program.

Kuhn’s definition has been applied beyond his original application to the history of physical science. Others believe that paradigms are the basis of normal science; indeed, the basis of established scientific tradition. In this view, the formation of a paradigm is a sign of maturity for a given science. The notion of a paradigm for programming was first expressed by Floyd (1979) as far as I have been able to discern.

We are going to apply the notion of paradigm to the investigation of programming languages and software architectures to determine how well we can solve different types of problems. The question we would like to answer takes the form: Is there a taxonomy within a particular domain that serves to organize the element of that domain? In programming languages, most computer scientists would answer “yes.” In software architectures, the answer is more likely a definite “maybe.” By the end of this book, we believe that you’ll see the answer for software architectures is “yes.”

## 1.2 SOFTWARE SOLVES PROBLEMS

How do we classify problems? Put another way, is there a typology of problems? If so, can we find common solutions that are widely applicable to many or all of the problems in the class? The recent emergence of patterns and frameworks suggests that there is such a typology and that we can find common solutions to them. Much work needs to be done in this area, but early results are very promising.

For a given problem class, we’d like to be able to create software that solves the problem or parts of it efficiently. There are two aspects to creating such software: the software’s architecture and the choice of programming language.

### 1.2.1 Software Architecture

Software architecture is the structure of the components of the solution. We decompose the problem into smaller pieces and attempt to find a solution for each piece. Sometimes, the solution for a piece requires further decomposition. When we have the solutions for each of the components, we have to integrate them and make them interoperate in order to have a piece of software, sometimes called an application, that solves the problem. Integration means the pieces fit together well. Interoperation means that they work together effectively to produce an answer.

There are many software architectures. Choosing the right one can be a difficult problem in itself. It is not clear yet what metrics are suitable for evaluating software architectures. We’ll address software architecture in more detail in Part III of this book.

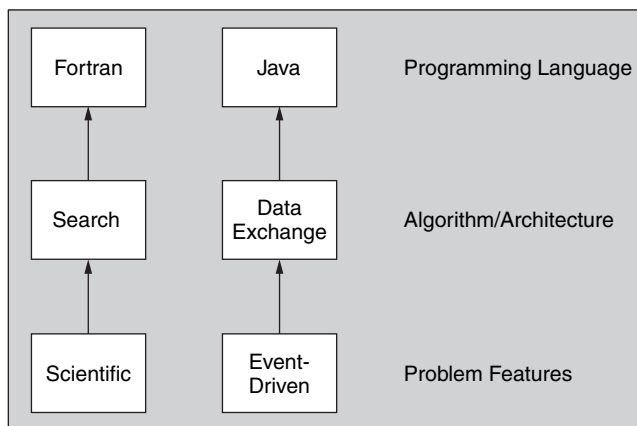
## 4 INTRODUCTION

**1.2.2 Choosing a Programming Language**

The second problem we face is choosing the best programming language to implement our software. There are different paradigms for programming languages that drive their syntax and semantics. There is no one programming language that is right for all problems. Alternatively, we do not know yet if there is one programming language that is best for a particular class of problems. In addition, we do not yet have the right set of metrics for deciding if this is so.

There have been a large number of programming languages. Jean Sammet (1967) counted at least 700, but most are now defunct. New programming languages continue to emerge, although less frequently than in the past. Recently, much work has focused on evolving and/or extending existing programming languages rather than creating new ones. Perhaps that is a sign of maturity in the field. George Michael (1980) once said that there will always be a Fortran, although we may not recognize today's language in it. Some features are being recognized as essential for writing good software—such as the evolution of object-oriented versions of older programming languages. Others, such as concurrency, support the way to create more efficient and better performing software.

So, faced with developing a solution to a particular problem, we have three tasks to perform (see Figure 1-1). First, we need to decide which set of features comprise our problem. This is the *requirements analysis* task. Second, we need to develop a suitable software architecture or algorithm for each subproblem and develop a software architecture for the overall problem. This is the *system design* task. Third, we need to choose a good, if not the best, programming language for each subproblem and implement the solution. This is the *implementation* task. That's not all there is to achieving a useful application, but that's the primary focus of our effort here.



**Figure 1-1.** Software design phases.

It is important to distinguish between the programming language, the paradigm on which it is based, and the algorithm that is used to solve the problem. Theoretically, it should be possible to write any algorithm in any general purpose language. However, some languages offer more support for particular software paradigms than others.

The problem of architecting a software solution to a problem is a complex one. It involves multiple tradeoffs that must be decided in order to yield a successful application that meets the requirements for solving the problems and the needs of the user(s) of the system. Inevitably, some combinations of trade-offs yield programming approaches that are clearly impractical for implementing an application. We have chosen to focus our attention in this book on structural paradigms for software systems.

### 1.3 DESIGNING AND DEVELOPING SOFTWARE

Software was originally called “soft” to distinguish it from the machine that executed it. Software is soft in a number of ways. It is not visible when it is running. It seems to change all the time, both when it is running and when it is being built. It is difficult to describe fully all aspects of software. In other words, there is no widely known way to describe the structure of a complete software system in the same way that the structure of a complete building, or an airplane, or a chemical plant is described. We can capture the static structure of a software system fairly well in many mature tools, but the dynamics continue to elude us. Nor do many of the well-known software analysis and design methodologies provide a notation for integrating the structure of the complete system with its dynamics.

We are interested in how to solve problems. In particular, we want to develop software that repeatedly can solve the same problem over and over with new data sets. In most cases, we understand the general approach to solving the problem. We may know, in fact, all the procedures and details. But, for all but the simplest problems, the thought of working through the algorithm(s) by pencil and paper is rather daunting.

Software allows us to capture the algorithms and detailed procedures in an electronically tangible form that permits almost effortless repetition (just push the button, they say). The hard part is designing the software, developing the software, and ensuring that it is correct. This book is about software design; we leave the latter topics for other volumes.

Today, most large software systems are too complex for a single individual to understand. Most of the software development we do these days we do with others because a single person hasn't the required skill or time to solve a complicated problem alone. One way to facilitate working together is to use and reuse concepts or ideas for which we share a common understanding.

When we write our first program, it often involves an intense amount of effort. The next time we performed a similar task, it is/may be a bit easier because we learned some things that we could reuse in our next project. As we develop more

## 6 INTRODUCTION

experience, we not only continue to learn new practices, but we refine and hone the practices that we have already learned. Some of them become rote because we have recognized them as “best practices” for programming and software development.

When we can capture and share the things we learn each time, we can reduce the need to rediscover what we’ve done before and share them with others who can also save time. This leads to a very important aspect of developing software that most of us don’t think much about but is basic to our success—the need for effective collaboration.

The technical literature is replete with books and technical papers on software design and the software design process. We do not intend to duplicate their efforts. We see software design as a three-pronged process: understanding the problem domain, designing the software architecture, and choosing the programming language. While addressing programming languages and problem taxonomies, our primary focus is on how to construct software.

### 1.3.1 Reusability

We have known for a long time that writing each new application “from scratch” is very expensive and time-consuming. There has long been an emphasis on reuse of software to mitigate some of the costs of developing new applications.

At the same time, developers felt that they could improve each succeeding application using the lessons learned from the preceding one, but that such use required

**Table 1-1. A reuse taxonomy**

Type of Reuse	Description
Ideas	The reuse of formal concepts such as general solutions to a class of problems
Artifacts	The reuse of software components (such as the Booch Ada components)
Procedures	The reuse of software processes and procedures
Vertical	The reuse of software within the same domain and even within the same application suite
Horizontal	The reuse of generic parts across multiple applications
Planned	The reuse of guidelines, development and testing processes, and metrics across multiple projects
Ad hoc	An informal practice in which components are selected from general libraries in an opportunistic fashion
Compositional	The reuse of existing components as the building blocks of new applications
Generative	The reuse of specifications and requirements to develop application or code generators
Black-box	The reuse of software components without any modification; usually linked together with glue code
White-box	The reuse of components by modification and adaptation

a complete rewrite of the application. By the 1990s, we realized as a community that successive applications could be built by reusing knowledge and software from earlier applications. This meant that succeeding applications could often be built faster than their predecessors.

### 1.3.2 Types of Reuse

At first, reuse referred just to the actual source code itself. Basili and Rombach (1988) expanded the definition to encompass everything associated with a software development project. Prieto-Diaz (1988) describes a taxonomy of eleven types of reuse as depicted in Table 1-1.

## 1.4 UNDERSTANDING PROBLEM PARADIGMS

A problem paradigm is a model for a class of problems that share a set of common characteristics. These characteristics serve to differentiate one problem type from another. Some problem types that you may have encountered in the past include search problems and classification problems (see Figure 1-2).

Over the past 40 years of computer science, we have identified classes of problems that share common characteristics. Problems such as search, enumeration, classification, data organization and manipulation, and sorting are encountered routinely in just about every application that is written.

When one thinks about it, there are relatively few problem structures that we encounter in computer science. However, we encounter them over and over in different contexts with different features and twists that make for interesting programming challenges.

Some of these problem paradigms have to do with the way data is structured and some with the type of processing to be applied to the data. Search, for example, is a recurring problem in many applications. But searching through a randomly ordered data set is different from searching through a known, sorted set represented as a vector or a B\*-tree.

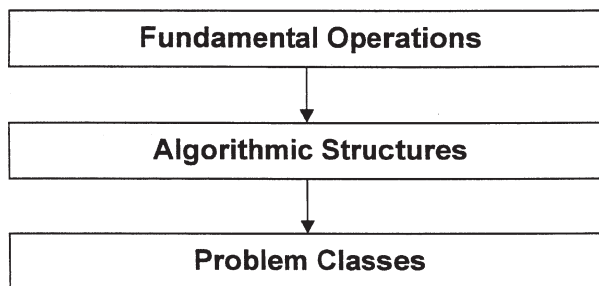


Figure 1-2. The problem paradigm.

## 8 INTRODUCTION

### 1.4.1 Fundamental Operations

At the lowest level of computation, there are some fundamental operations that must be performed to solve any problem, no matter how they are syntactically represented. Among these are assignment of values to variables, decision making, and sequencing through instructions as shown by Bohm and Jacopini (1966). As one considers more complex programs, one realizes that other operations become fundamental as well. We would argue strongly that recursion, procedure calls, concurrency, and exception handling—to name a few—should also be considered as fundamental operations.

### 1.4.2 Algorithms and Architectures

An *algorithm* is a specification for how to solve a particular problem. Algorithms are ordered sets of unambiguous, executable steps that define a terminating process. The key idea is that an algorithm is deterministic. With the proper inputs to the algorithm, repeated execution of the algorithm will return the same result. In addition, an algorithm must complete (e.g., yield a result) in a finite amount of time.

Algorithms can be written in any language from English to programming languages. Recipes for how to bake a cake or make chicken cordon bleu are algorithms. These two expressions represent the same algorithm:

- The Fahrenheit temperature is nine-fifths of the Celsius temperature plus 32.
- $\text{Fahrenheit} := (9/5) * \text{Celsius} + 32$

Algorithms are translated into programs that are executed on computers. When we study algorithms, we are concerned about how much time and how much space they take to compute the result. We can derive a hypothetical result through careful analysis of the algorithm, but we are often concerned with how the algorithm performs when it is translated into a program using a specific programming language, running under a specific operating system, on a specific computer architecture.

### 1.4.3 Problem Classes

A *problem class* is a set of problems that are similar in nature although they may be dissimilar in their origin (e.g., the field from which they emerge). For example, checking a book or a videotape out of a public library is similar to checking a software module out of a system library. Both systems have similar fundamental operations and algorithms that characterize them. Recognizing the set of operations and the set of algorithms leads us to describe a problem class, for example, a set of operations that routinely occur in solving a particular type of problem. Some of the types of problems that we routinely encounter are search, sort, classification, enumeration, and graphics.

### 1.4.4 Summary

Several insights arise from this approach. First, problem paradigms are nothing magical. We have been writing programs to solve problems for over fifty years. During this period, some accepted standard solutions have emerged for given problem classes. Within the past ten years, this has been captured in the subdisciplines of patterns and frameworks, which represent problem paradigms. Second, if we can describe a problem paradigm, we can implement it on a target architecture, albeit not efficiently sometimes. Third, there is no theoretical reason why we cannot combine multiple paradigms to solve more complex problems. However, practicality is an entirely different matter. Some argue that the use of multiple patterns is just such an approach. I agree. Fourth, the target architecture has a strong influence on the way a problem paradigm is implemented. There must be a synergy between the problem paradigm and the computer architecture to obtain efficiency in execution. Fifth, any programming language can be used to implement any problem paradigm; some will do it more efficiently than others. Finally, there are numerous problem paradigms, which suggests some way of organizing them into a hierarchy or mesh-like structure according to certain properties and characteristics.

## 1.5 OVERVIEW OF BOOK

This was a difficult book to write. It mixes concepts from three different areas—programming languages, software architecture, and basic computer science—in an attempt to develop an interdisciplinary approach to creating software. Its interdisciplinary nature created a problem in itself: What's the best way to organize the material to make it both understandable and usable?

After some reflection, it seemed there was a natural hierarchy emerging that ran from programming languages and data constructs through more complex structures. This hierarchy is described in Chapter 2.

I have chosen to address each of these elements in a separate part within the book. Each part is composed of small chapters that address particular topics. Each chapter in itself is worthy of a whole book, but my intent here is to show how they are all interrelated. In each chapter, I suggest additional references that will allow the reader to delve more deeply into the material.

This book is divided into several introductory chapters, multiple parts, a bibliography, and a glossary.

Chapter 1, this chapter, provides a gentle introduction to the book and the topics of discourse.

Chapter 2 provides an overview of the paradigms that we will discuss in the remainder of the book. It presents a limited catalog of the types of problems that we solve with programming languages. It briefly describes the hierarchy of architectural paradigms: design patterns, components, software architectures, and frameworks.

## 10 INTRODUCTION

The remainder of the book is divided into four parts, each of which consists of multiple chapters. Each part focuses on a specific paradigm. Each chapter focuses on a specific topic.

### 1.6 CONVENTIONS

In this book, I have used a number of conventions to simplify the reading.

I use the masculine pronoun when writing in the third person although I mean both male and female readers.

URLs for the World Wide Web are included in this text. They have been verified to be accessible as of the publication date of this book. If you find a URL that is not accessible, please notify the author. However, you should be able to search on words or phrases of that section to find other URLs pertaining to the same material.

In some chapters, I have added references for further reading for source material that is not referenced directly in the book.

I have included exercises at the end of each chapter. I hope that the answers to these exercises are not obvious: that is, you will have to do further investigation and analysis beyond the content of the chapter in order to arrive at a satisfactory answer. For many exercises, there are no right answers.

### 1.7 EXERCISES

I had thought about putting some summary exercises after the last chapter but eventually decided that they should be right up front where you can read and ponder them, and, hopefully, they will stimulate some comparative analysis as you read through the remainder of the book.

- 1.1. In Section 1.3, review Prieto-Diaz's reuse taxonomy. Then, build a table with the reuse types versus the paradigms and fill in the cells with justifying information for why a particular paradigm supports a particular reuse type.
- 1.2. In Section 1.3, I assert that there are large software applications today that are too complex for any single individual to understand. Indeed, this has been said about OS/MVS and succeeding releases since the early 1980s and Unix and Microsoft Windows more recently. How big, according to some unit measurement, do you think a software system has to be before it cannot be comprehended by a single individual?
- 1.3. Section 1.4.3 discusses the concept of problem classes. There are some well-known types of problems in computer science. Develop a taxonomy of problem classes for use in later exercises. [Research Problem]